**Gen-Z Manageability Simulation and Proof of Concept**

**Submitted To**

**Deji Akinwande**

**Jon Hass**

**Dell USA**

**Prepared By**

**Albert Bautista**

**Cameron Clark**

**Wesley Klock**

**Jake Morrissey**

**George Zhang**

**EE464 Senior Design Project**

**Electrical and Computer Engineering Department**

**University of Texas at Austin**

**Spring 2020**

**CONTENTS**

**TABLES iv**

**FIGURES v**

**EXECUTIVE SUMMARY vi**

**1.0 INTRODUCTION 1**

**2.0 DESIGN PROBLEM STATEMENT 1**

**2.1 Design Problem Description 1**

**2.2 Design Problem Specifications 2**

**2.3 Design Problem Deliverables 4**

**3.0 DESIGN PROBLEM SOLUTION 5**

**3.1 Subsystem #1: Management Console-Frontend 7**

**3.2 Subsystem #2: Management Console-Backend 8**

**3.3 Subsystem #3: Redfish Simulation 9**

**4.0 DESIGN IMPLEMENTATION 11**

**4.1 Design Implementation: Management Console-Frontend 12**

**4.2 Design Implementation: Management Console-Backend 13**

**4.3 Design Implementation: Redfish Simulation 14**

**5.0 TEST AND EVALUATION 15**

**5.1 2D Map Display Testing 16**

**5.1.1 *Method* 16**

**5.1.2 *Results* 18**

**5.1.3 *Analysis* 18**

**5.1.4 *Recommendations* 19**

**6.0 TIME AND COST CONSIDERATIONS 19**

**6.1 Learning and Extending the Redfish Simulation 19**

**6.1.1 *Learning the Redfish Specification* 20**

**6.1.2 *Understanding and extending the Redfish Interface Emulator* 20**

**CONTENTS (*Continued*)**

**6.2 Narrowing Overall Functionality 20**

**7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN 21**

**8.0 RECOMMENDATIONS 22**

**9.0 CONCLUSIONS 23**

**REFERENCES 24**

**APPENDIX A – [RELEVANT STANDARDS] A-1**

**APPENDIX B - [FRONTEND FIGURES] B-1**

**TABLES**

1 Functional Requirements 3

**FIGURES**

1 Gen-Z Memory-Centric Architecture A-2

2 System Block Diagram 5

3 Frontend Component Table B-2

4 Frontend Redfish API Table B-3

5 Redfish Service for Gen-Z Attached Enclosure 10

6 Expected Redfish Simulation for 2D Map Display Testing 17

7 Actual Frontend Display for 2D Map Display Testing 18

**EXECUTIVE SUMMARY**

To achieve the goal of providing a proof of concept for manageability of the Gen-Z interconnect, we created a Management Web Console connected to Redfish simulators that represent Gen-Z devices. Through the Redfish Simulation, we simulate the device data and interface that future Gen-Z manageability products will utilize. Through the Management Console, we provide a User Interface that unifies the simulated Gen-Z device data to display to the user in a human readable fashion that serves as a manageability proof of concept.

The purpose of this paper is to provide a complete and comprehensive overview of the design of the Gen-Z simulators and our manageability console design. This report provides a brief overview of the Gen-Z architecture and explains every element of our management console proof of concept senior design project.

The design problem is a proof of concept for a Gen-Z manageability console. The Gen-Z manageability platform must allow for an IT user to view a Gen-Z system and detect any errors in the devices which we will simulate. Gen-Z technology is trying to solve the problem of dependent memory causing performance hits in composable systems. In order to successfully complete the proof of concept Gen-Z management project, it must be able to show on the management console the connected components with Gen-Z information according to the Gen-Z specifications. The project must deliver a functional, interactive simulation of a set of devices simulating Gen-Z technology in a human readable format displayed on a website.

In order to help with the development of management tools for Gen-Z systems, we developed a management console and simulation of a Gen-Z system. The solution contains 3 sub-systems: a console frontend, a console backend with an attached MongoDB database, and the simulators. The frontend has a component table, where you can individually view each component and navigate through the simulation easily. It also has a Gen-Z relationship mapping, which shows how the components are connected over the Gen-Z fabric. Data from the simulators can be easily added or deleted by providing the path to a simulator in the Redfish API table. The backend acts as an intermediary between the simulators and the frontend, allowing for asynchronous communication and monitoring of changes in the simulator state. The data is stored and timestamped in a MongoDB database. The backend queries these simulators using HTTP endpoint requests over LAN. The requests and the endpoint data adhere to the Redfish specification. For the simulators, there are multiple simulators running that represent separate devices and enclosures. Devices we are simulating include Gen-Z switches, fabric adapters, media controllers, memory chunks, compute systems, and PCIe devices such as GPUs and comms devices. Zone information is stored in its own simulator and is represented by a colored circle around the nodes in the Gen-Z relationship mapping view.

Our testing was relatively light with a single end-to-end test between the simulation and 2D map. The system was tested using full end-to-end testing to verify the information on the simulators was correct in the backend and frontend. The test serves to evaluate the expected user experience based on the configuration from the simulation data with the data shown in the frontend display

In focusing our design, we used the static simulation and the Gen-Z relationship mapping due to time considerations. We had to put in the time to understand how the Redfish specifications and the existing codebase, the Redfish Interface Emulator, worked before we could make the changes we wanted. We narrowed down the overall functionality of our project from having a full set of management features initiated from the management console such as Gen-Z composable systems assignment and dynamic status updates to focus on the 2D mapping showing connected devices over the fabric.

Our design took into consideration privacy and security concerns, though we didn’t have much time to implement those features. In the development of our design, security and safety concerns around digital security are important due to the nature of our management proof of concept contributing to the overall ecosystem of IT data centers. We considered looking at legislation around the space of information security since our design was involved in information security.

Moving forward, we recommend that someone advancing this project should move the simulators to micro services and host the frontend and backend on servers to increase accessibility. Additionally, they should add functionality for dynamic behavior of the simulators and add a physical mapping view on the frontend.

Ultimately, our project provides a proof of concept for management of a Gen-Z system by providing human readable information of components and connections in an accurately modelled Gen-Z network.

**1.0 INTRODUCTION**

Our main goal was to develop a manageability simulation to model a software platform that would accommodate Gen-Z components, connections, and use-cases while maintaining a user-friendly design. This provides a proof of concept simulation to track, visualize, and interact with the Gen-Z fabric efficiently. Gen-Z fabric serves as an efficient, agile, and adaptable interconnect between memory and devices to enhance the speed and efficiency in existing computer architectures.

The purpose of this paper is to provide a complete and comprehensive overview of the design of the Gen-Z simulators and our manageability console design. This includes a detailed discussion of our design problem, our specific implementation, the testing and evaluation, and future recommendations for the project.

**2.0 DESIGN PROBLEM**

The design problem is a proof of concept for a Gen-Z manageability console. In order to manage Gen-Z devices, the information must be easily understood by the end-user and displayed in a logical manner. Additionally, we will have to model a system with Gen-Z components to create a dataset from which to display.

**2.1 Design Problem Description**

The Gen-Z manageability platform must allow for an IT user to view a Gen-Z system and detect any errors in the devices which we will simulate. Gen-Z is a relatively new technology aimed to provide easy data-access among multiple components. There has been a lot of development in the creation of Gen-Z devices, but there is a need for a management platform that displays the underlying structure and status of the network. For example, this will allow an IT administrator to view the status of a set of Gen-Z devices that are connected to the management platform and see which devices are on, which are not working, which need updates, and other pertinent information. These tasks that the IT administrator executes to maintain the devices are commonly referred to as workflows.

Gen-Z technology is trying to solve the problem of dependent memory causing performance hits in composable systems. Gen-Z technology can be mapped into any processor memory management unit to translate operations to move up to 232 bytes of data. The Gen-Z Consortium, a conglomeration of technology companies, is developing the standards of Gen-Z technology [1]. Since it is still being developed, the technology remains relatively flexible. This new standard introduces a different architecture from the old CPU-centric architecture with dependent memory into one that is memory centric as shown in Figure 1 for Appendix A.

To create an example system with Gen-Z technology, we must use simulations adhering to Redfish specifications. Redfish is a RESTful API that defines management protocols for a variety of systems or devices. These devices range from a “stand-alone server with various components” to “larger-scale cloud environments.” This will allow us to interface with the Gen-Z devices and provide them commands [2].

**2.2 Project Specifications**

In order to successfully complete the proof of concept Gen-Z management project, it must be able to show on the management console the connected components with Gen-Z information according to the Gen-Z specifications. The simulation needs to be able to send these statuses and configurations of all the connected devices to the management console. The functional requirements are shown below in Table 1 for ease of readability.

**Table 1. Functional Requirements**

|  |  |
| --- | --- |
| **Functional Subject** | **Requirement** |
| Simulated Devices | Management console should update and display according to the new devices that the API endpoint simulates |
| Component Information | The component information will be transformed into an easier to read format with easy ways to traverse the links up or down the simulation. |
| Gen-Z Specific Information from Simulation | The console allows for view the status of all connected devices to the system, as well as nodes and configurations relevant to managing the system like Gen-Z id’s and connections to switches or fabric adapters |
| Simulation representing Gen-Z Device | The simulated device should represent devices such as memory, peripheral component interconnect express (PCIe) devices, fabric adapters, and switches. |

The management console will interact with the newly simulated devices to compare and update the reported status to the user display. Additionally, the frontend management system will have a user interface that will display component information. The component information will be transformed into an easier to read format with easy ways to traverse the links up or down the simulation. All links within a component will be named and shown with a clickable link to go to the linked component. Each simulated device will return a response detailing current component status including, but not limited to, Gen-Z specific data. For example, the response will return its identifier and how it is connected to other components like a Gen-Z switch or fabric adapter.

In order to effectively monitor the status and health of devices, all information should be accessible, readable, and match expected user interface designs like clickable links and color-coding statuses. Displaying and detecting the components or devices from the simulation should be within a reasonable timeframe in the scale of seconds. The system should not suffer large performance hits if there are less than forty nodes in the Gen-Z Relationship Mapping on the frontend. It should also run smoothly with less than 10 simulators running. We made this determination by looking at mockup data that had a very simple system with less than 10 devices. We decided to extend the system but not make it too complex, so we set reasonable upper limits that let us create a useful Gen-Z system. The user interface, or UI, will be able to view the status of all connected devices to the system, as well as nodes and configurations relevant to managing the system like Gen-Z id’s and connections to switches or fabric adapters. For the proof of concept, the simulation should be able to simulate the common devices that one would encounter in a typical server infrastructure like memory, peripheral component interconnect express (PCIe) devices, fabric adapters, and switches.

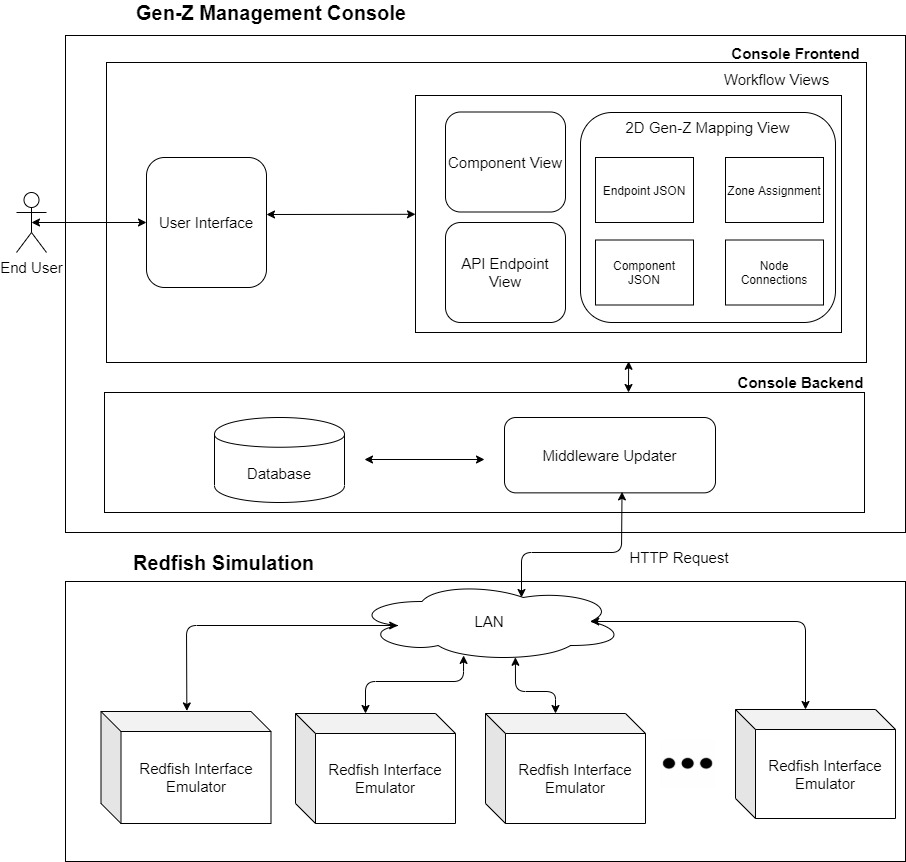
**2.3 Project Deliverables**

The project must deliver a functional, interactive simulation of a set of devices simulating Gen-Z technology in a human readable format displayed on a website. The design will consist of creating a management console separated into frontend and backend. The console frontend allows the user to see the components in a human-readable manner. The console backend will handle interacting with the simulation. It will query for any changes and store the data into a database. The information of the components will be held in simulations of the devices in JSON files. The information displayed on the management console must be reliable, up-to-date, and accurate.

The user will be able to see the devices that are being simulated and all the fabric connections for all the devices. The information displayed will be accurate and recent. This means that the database in the console backend will have to be up to date and provide accurate information to the frontend regularly in the scale of seconds. The frontend will have to understand how the devices are connected over the Gen-Z fabric and display that information properly. Additionally, latency between changes to a device and the display on the frontend cannot be too slow due to the need for reliability.

**3.0 DESIGN SOLUTION**

In order to meet our specifications, we chose to add a backend database, use a particular software stack, and build off of existing data and code for the simulators. The system is made up of three distinct subsystems: The Gen-Z Management Web Console, which includes two subsystems in the frontend user display and the backend database of cached information about the system, and the Redfish Gen-Z Simulation utilizing modified Redfish Interface Emulators. We built our simulators from a codebase called Redfish Interface Emulator, which is a Python Flask application developed by the DMTF, an organization that defines manageability standards, that uses Redfish to represent IT devices. The system block diagram for our design is shown in Figure 2 below.



**Figure 2. System Block Diagram**

The Gen-Z Management Web Console is the interface that the end user interacts with to see the Gen-Z connected components, how the components are connected with each other over the Gen-Z fabric, and the status of each component connected to the Gen-Z system. It also allows for the user to add or remove a Redfish API port that a simulator is run on to add or delete data from a simulator. The console is divided into two subsystems: the frontend and the backend. The frontend unifies the data received from the simulation and presents the data in a human readable format. To meet this specification, the frontend has two views: a component specific view to see the data for a specific device component, and a 2D relationship map that displays zone information and how the components are connected to the Gen-Z fabric through switches and fabric adapters. The backend holds the central database that polls for any updates using HTTP requests through two channels, one communicating with the frontend and the other communicating with the simulation. The backend regularly checks for changes from the simulation to store in the centralized database and ultimately updates the management console to reflect the change. The backend also processes the incoming simulation data to note metadata in the simulation for the frontend to use. This includes Gen-Z nodes for the frontend 2D map to display. The simulation is made up of HTTP endpoints accessible through Local Area Network (LAN) that holds data according to the Redfish specification that the backend accesses and stores into its database.

The frontend and backend are using a MEAN stack (MongoDB, Express.js, Angular, Node.js). The frontend is composed of an Angular framework which will call the backend composed of Node.js and Express.js. The backend will also connect to a MongoDB database which stores the latest simulator data.

Finally, the Redfish simulation utilizes Python Flask to run instances of the modified DMTF Redfish Interface Emulator to represent the simulated devices. The dataset is represented through JSON files that are accessible through HTTP requests to the simulator. Through this system, we can demonstrate a system in which a server administrator can complete workflows with the efficient and usable interface to manage a Gen-Z supported server.

**3.1 Subsystem #1: Management Console-Frontend Display**

The Frontend Display is composed of several views that serve as a different way of viewing the simulation information. Initially, we realized we needed at least two views: An Overview Dashboard that helps the user access basic status information and component identification for all individual components and the Device Details View that displays all available information for a single component. Additionally, we wanted to include a view to show the Gen-Z relationships of devices. The Gen-Z Relationship View will help the user grasp the complex nature of relationships between switches, memory units, and processors within the Gen-Z fabric.

The Component View is a tabular view that displays basic information about each of the devices within our system including health status, memory capacity, and networking protocols for each component. The Component View is shown in Figure 3 of Appendix B. In this dashboard, the user can search by the name of the device or component. If the user needs a more detailed explanation of the status of a device, they can click on the device’s row, which will navigate them to the Device Details View. We specifically chose this overview format because many existing network management systems have similar layouts, which will improve usability for the network system managers that are likely to be using this system. A variety of workflows can be accomplished from this view, including maintenance of the systems, which requires viewing health statuses and physically adding and removing parts.

The Device Details View will help the user look for specific information on a device. To help understand what would be most important to display, we utilized our prior art research from the Network Device User Interface [3]. The detailed device view focuses on similar aspects of configuration and status within the larger network. As a result, the Device Details View will have an initial generalized layout that is modified for each type of device. By clicking a connected device, a user will be able to navigate to a Device Details View for that device. The hierarchical nature of the device structure in the simulators will be displayed at the top of the page with a path to the device from the system level. This will allow users to easily navigate between high level devices to the more specific devices they contain. Links for connected devices within a Device Details View are listed on the right with the name of the device. This will allow for easy navigation of nearby components.

When designing our Fabric Relationship View, we took inspiration from our prior art research from the Mellanox NEO GUI. The Mellanox NEO GUI had UI elements very similar to what we envisioned for this relationship focused view. Specifically, Mellanox’s network map interface was a core source of inspiration for our design. The graphical representation of the system displayed each component as an icon that reflected the type of device it was by the label and the color. These icons acted as nodes in the graph, while the edges or lines between devices represented valid connections [4]. Our view will use a similar style but will instead focus on Gen-Z relationships instead of purely network ones. This helps the user access the less hierarchical nature of Gen-Z relationships and see how devices are connected through switches, ports, and fabric adapters. Rings will be used to show the groupings of Gen-Z devices represented as zones so that the adding and removing of devices from these groups can also be visualized. For each device, the user can access identification, health information, and all the other information found in the Device Details view by clicking on the icon.

**3.2 Subsystem #2: Management Console-Backend Database**

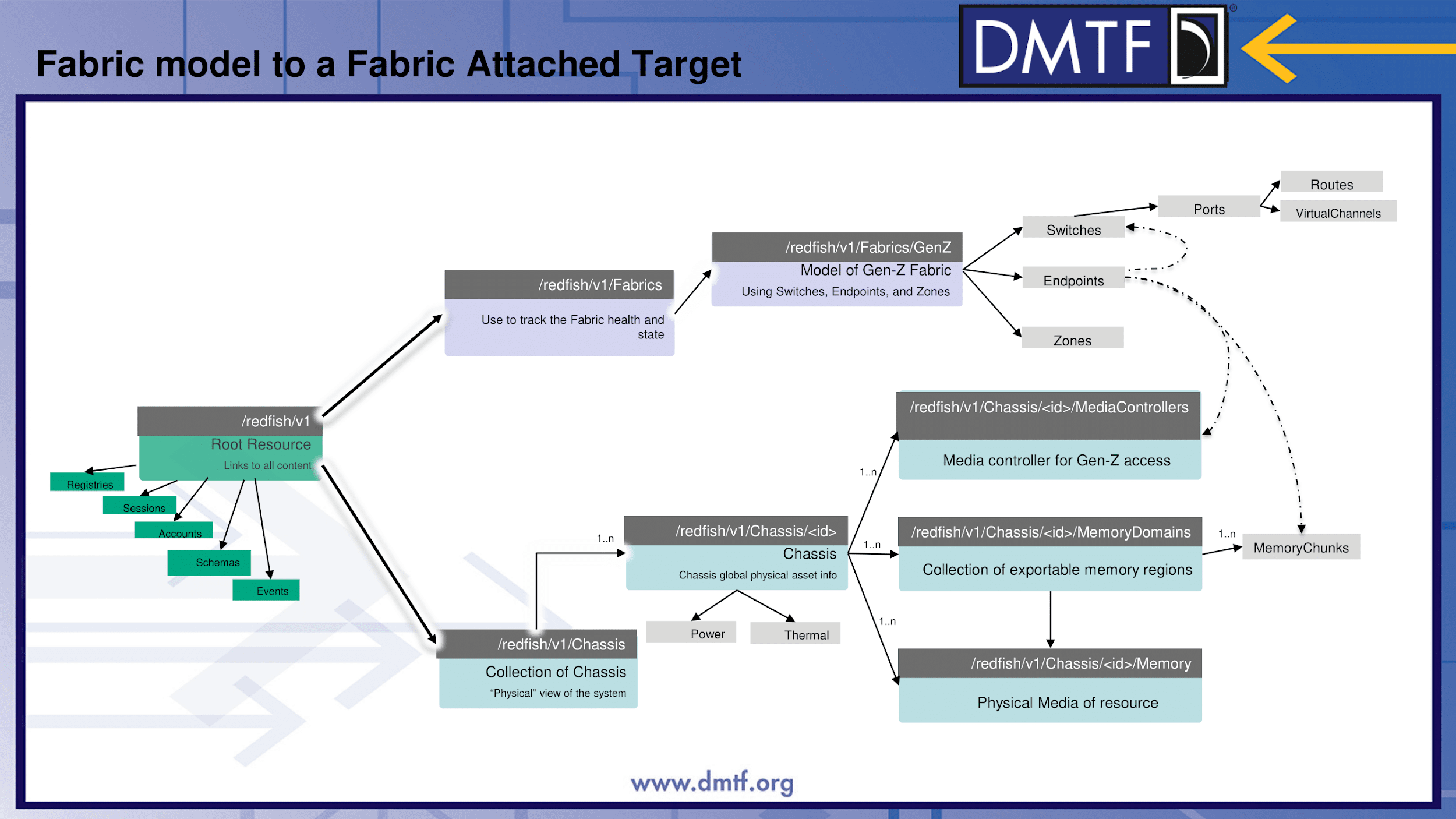
The backend is a program that handles HTTP requests from the frontend to display or update the data in the database. The backend also implements a scheduled task that pulls from the simulator systems to update the central database. The database stores the most recent simulator data for the frontend to display. This way the most recent data is known if a simulator crashes. The backend is also designed to only need the Redfish API domain of the simulator to update the database with new data. The backend can recursively discover all the devices on a path, so only the top-level Redfish API domain is needed. The frontend user can add in a domain, and the backend will update and add new data accordingly to be displayed. How this looks on the frontend is shown in Figure 4 of Appendix B.

Though the backend updating could be integrated to the frontend, we chose to separate the process into its own subsystem to allow for modularity of the system and testability of the system. This is where we can easily pinpoint and test the latency of the system. The backend solution allows for the updates from either subsystem to be asynchronous to the other system and makes for an efficient implementation to communicate the necessary data. Since the Redfish API utilizes HTTP requests, both channels from the backend to the other two subsystems can utilize the same protocol to be able to test the single system that can work for both ends of the solution.

**3.3 Subsystem #3: Redfish Simulation**

The Redfish Simulation is composed of multiple instances of the simulator running a modified DMTF Redfish Interface Emulator. Based on our mentor’s input, we looked at several Redfish simulators made by DMTF and found that a codebase called the Redfish Interface Emulator (RIE) was the most fleshed out and flexible option. Another important reason it was picked was because of team members’ preference with Python as opposed to the other applications written in Go. The RIE is an application designed to emulate a Redfish resource or service as an HTTP endpoint accessible through the local network on the machine that the emulator is running on. The Redfish service is the logical view of what a service processor for a server rack would see. The system is composed of multiple components ranging from the networking fabric of Gen-Z to physical components like PCIe devices and memory.

The Redfish Interface Emulator is a Python Flask application that allows the simulation of the HTTP interface that one would receive from a Redfish service. The Redfish service encapsulates a varying range of configurations. An example of this configuration for Gen-Z could be a Gen-Z connected enclosure holding memory. It would be represented in a file structure as seen in Figure 5 below.



**Figure 5: Redfish Service for Gen-Z Attached Enclosure**

In focusing our project, we went with the direction of static emulation of the system due to the need for prototyping proof of concepts for Gen-Z components because certain fields needed to be added and changed in accordance with our Industry mentor’s guidance. Dynamic emulation for the components required more time for each resource along the URI. In addition to narrowing the simulation, we have chosen to focus on simulating the Gen-Z specific data as opposed to a holistic view of the system with information not pertinent to Gen-Z. Some examples of information not related to Gen-Z would be temperature data or fans in an enclosure. These changes in the implementation of the simulation is to help focus the project as a proof of concept specifically focused on Gen-Z management.

There are multiple simulators running simultaneously over local connections. For setting up the RIE instances, we are running the simulators locally for the purposes of the proof of concept. The simulators are connected by LAN and accessible to the console backend system. We created a Python script to allow for easy booting up of multiple simulators. All that is needed is the top-level file path and the port that it should be run on. This allows for a more accurate simulation where devices in a certain simulator could change or crash without affecting devices that aren’t connected to it. In total, we have eight simulators running simultaneously on our design.

Changes we made to the simulator include support for remote links and making memory chunks their own endpoints. If a simulator contains information that links to a path that is running on a different simulator, that link now contains the port number in the data to allow for easier access and management by the frontend. Before, it was assumed that any links would link to data running on the same port. The second change was creating endpoint information for memory chunks. They had been represented within the media controller’s endpoint. However, that created an edge case for display on the frontend, because we were using endpoints to create the nodes for the Gen-Z Relationship Map. To make it easier for the frontend team to display the information, we created endpoints specifically for the memory chunks.

**4.0 DESIGN IMPLEMENTATION**

Our design decisions included the displays we wanted for the frontend, the decision to create a backend for a middle layer between the frontend and simulators, and the alterations we made to the simulators given our time constraints. For the frontend, we contended originally with several component views from health status to a physical connection visual map for our solution. In consultation with our industry mentor Jon Hass, we focused on the 2D Gen-Z mapping view and Component Table. For the backend, we made the key decision to build out a middle layer between the web console and the simulation. We changed the database format to work with the given simulator Redfish data and created metadata from the simulation data for the frontend mapping to work. For the simulation, we had to understand and modify the DMTF Redfish Interface Emulator within the given time constraints. The key decisions for the simulation was choosing whether to start from scratch or work with a simulation program written by the DMTF, developing simple static JSON mockups as opposed to dynamic Python scripts, and setting aside changing the mockup data during runtime to focus on building out Gen-Z connections to display on the 2D map.

**4.1 Design Implementation: Management Console-Frontend**

The frontend’s purpose is to display all the connections and data from the simulators and to display the Gen-Z mapping representation of the simulators. The purpose was refined over time through discussions with our industry mentor. The original idea was to create multiple displays and each display would display a different field in the components. However, after further discussions as a team and with our faculty mentor we refined our design to make one master display that displays all the components in a table, highlighting the most important fields such as health status. The table would include the functionality to click into a specific component to find all the available information about the component and also the ability to navigate to its connected components. Later on, in development, we also decided to add a second display that visually displays the relationships between components within the Gen-Z fabric in a 2D graph.

In order to allow efficient use of code and also easily support additional displays in the frontend, we decided to use the MEAN stack. The MEAN stack is a free and open-source JavaScript software stack used for building dynamic and easily scalable web applications. The MEAN stack is made of MongoDB for the database, Express.js for the backend to handle API requests, Angular for the frontend display, and Node.js for the backend runtime environment. A significant reason why we decided to use the MEAN stack is because of the large amount of information about the MEAN stack and Angular. Angular allows us to efficiently use code for multiple similar displays. Angular is also scalable allowing us to add multiple displays as our design changed over time without affecting our current design.

After implementing the MEAN stack and designing the component table view with the navigable components, the next step after discussion with our faculty mentor was to implement a Gen-Z relationship view. This view’s purpose is to show the relationships between each component in the Gen-Z fabric in a 2D graph. The first step was to find a library that will support a graph with a dynamic number of nodes and edges connecting nodes to each other. It was also important to find a graph that has several design options in terms of coloring and showing the groupings of certain nodes. The library that fit our requirements the best was the D3.js library. The D3.js graph allowed us to actually implement our idea on how to visually show the Gen-Z relationships.

All assets of the frontend were refined and tested through discussions and demos as a team and with our industry mentor. Through user testing and demonstrations, the frontend design was refined to our liking. Discussions became an important part to our design of the frontend due to the success of the frontend being strongly tied to the user experience. If the frontend is not intuitive and easily understandable, then the frontend does not successfully complete its purpose. Therefore, many discussions were required to achieve our end product.

**4.2 Design Implementation: Management Console-Backend**

Our goal for the backend was to handle the http requests from the frontend to access the database as well as handle updating the database with the new simulator data. The reason why we do not have the frontend directly pull data from the simulators is to support many users at once. Having all users request data from the same simulators would be very taxing and slow. The existence of the backend also supports error cases where simulators crash. It is necessary to have a caching system, so data is not lost when simulators break. Having the latest state of the simulators, including the last state before crashing, is important information for simulator management. The backend is made up of a MongoDB database, Express.js, and Node.js.

MongoDB database was an important asset to our web console stack because MongoDB is a NoSQL database and uses JSON-like schemas. This was very useful because the simulators did not have any consistent formatting. Certain components had completely different JSON schema from each other, necessitating a database to support any object as a field. MongoDB meets these specifications.

Even though MongoDB does not have a fixed format for acceptable data, we still needed to decide how to categorize our data in the database into collections. The core requirement we were trying to solve was a way for the user to only need to supply the API endpoint of the simulator to populate all the data at that endpoint. In this way, there is no manual adding of data by the user, and the user only needs to supply the minimum requirements to add new data. To achieve this functionality, we decided to create three collections in the database and three cron jobs. The cron jobs, a piece of code that repeats at a set rate, will run every ten seconds and will be responsible for updating and adding new data. The three collections are a collection of API endpoints pointing to each simulator, a collection of links found in each component, and a collection of the components storing all their data. The first cron job looks for new API endpoints that the user may have added. When a new API endpoint is found it will be added to the collection of links. The second cron job is responsible for looping through all the links in the collection of links, sending an HTTP request for the data associated with that link, and adding or updating that components data in the collection of components. The third and final cron job is responsible for finding new links embedded inside the data of the components. The cron job will loop through all the data in the component collection and add any links found in the component that do not already exist in the collection of links. In this design, the user will only need to supply the API endpoint and the backend will recursively search for new links in the collection of components and update or add new data accordingly.

**4.3 Design Implementation: Redfish Simulation**

We chose to modify the Redfish Interface Emulator, work with static configurations, and build out an example Gen-Z system for the frontend to display. For the simulation codebase options, we could build out our own simulation or work and modify an existing Redfish simulation program created by DMTF. There were two codebases by the DMTF, one in Python and the other in Golang. We chose to work with the Python codebase from DMTF called the Redfish Interface Emulator (RIE) as the team was more familiar with Python.

Next, we made the key decision to only work with the static RIE configuration for the interest of time due to the work required for the dynamic configuration. The benefits of dynamic emulation include being able to do GET, POST, PATCH, and DELETE requests instead of just GET requests like the static emulation. Dynamic emulation would allow easy changing of data while the simulator is running. However, dynamic emulation requires writing scripts for each path for a Redfish service and modifying the codebase for each endpoint. The initial codebase did not fully run with dynamic emulation for the mockup data we had, so it would require updating it before we even added our own extensions to the mockup data. Dynamic emulation would require time in both creating the scripts to work as intended and testing each script for every path in the Redfish service. On the other hand, static emulation would take the JSON files of a specific folder and would represent the JSON output of each path in the Redfish service. The most time-intensive part of the static emulation would only be altering and adding the JSON files. Since our project was a proof of concept for Gen-Z management as opposed to direct emulation of the specific device data, we considered the cost of time to develop the simulation and chose to go with static emulation. The time working with building out RIE scripts was better spent directly working with the mockup data. We could fully flesh out an example Gen-Z system to represent on the frontend if we chose the static configuration.

The final key decision that we made was to focus on developing a good set of mockups to represent a wide range of Gen-Z connections and configurations as opposed to working on changing the JSON mockups while running to simulate expected behaviors like machine startup. Initially, the simulation was planned to simulate changes like machine startup behavior that would be reported to the web console to allow for startup oversight. We modified the RIE to allow for the changes to take effect through the restart of the RIE instance on any mockup changes. Additionally, we wrote classes to simulate the managers that would initiate the changes to a specific mockup. As the project developed through feedback with our industry mentor Jon Hass, we focused more towards the functionality of the 2D mapping as opposed to displaying dynamic changes to the simulation data. Therefore, we chose to sideline the dynamic modification of the mockups to focus more on a diverse set of mockups of Gen-Z configurations. We are running eight total simulators representing various devices and enclosures. They are connected in various ways over a fabric like a potential Gen-Z system would have. We have various fabric adapters, PCIe devices like GPUs and comms devices, media controllers, memory chunks, zone information, and three Gen-Z switches. This system was able to be developed and tested due to our decision to use static emulation.

**5.0 TEST AND EVALUATION**

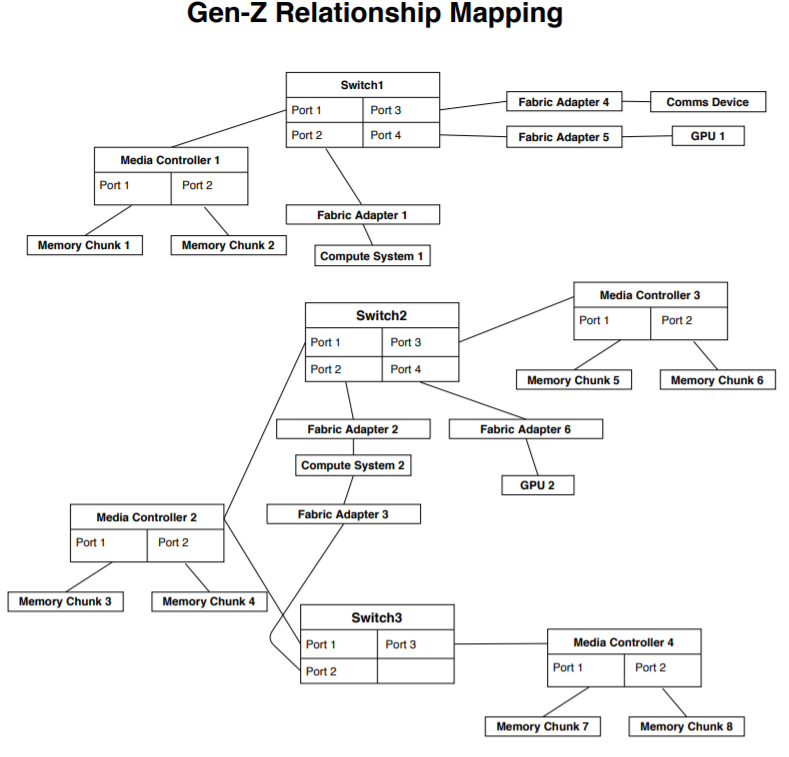
Our testing was relatively light with a single end-to-end test between the simulation and 2D map. We built our project based on weekly feedback from our technical mentor. Since we frequently changed scope and took into consideration setbacks in learning new technologies, there was a lack of emphasis on building tests for our software. We programmed feedback in our software to ensure we could double check any possible issues. For example, the frontend will display if there are any dead links for a specific component. A dead link means that a link exists in the JSON file, but there is not a valid HTTP response for that link. The cause could be it is data we are not representing due to our focus, or it could be an error like a simulator not running or a typo. For the final solution, we made an overall end-to-end test comparing the simulation with the frontend 2D map. This would allow us to check that all the data in the simulators was being displayed properly on the frontend. The map became the culmination of the proof of concept goal in which the representation unifies Gen-Z device data from the simulation that would be displayed in a simple interface. The frontend map needed to accurately display what was on the simulators.

**5.1 2D Map Display Testing**

The 2D Map display unifies data from several endpoints in the simulation to show specific port connections for Gen-Z switches, what Gen-Z zones specific components belong to, and a variety of devices and configurations of common IT devices like memory, PCIe devices, etc. The testing serves to validate the simulation data with the frontend 2D map.

***5.1.1 Method***

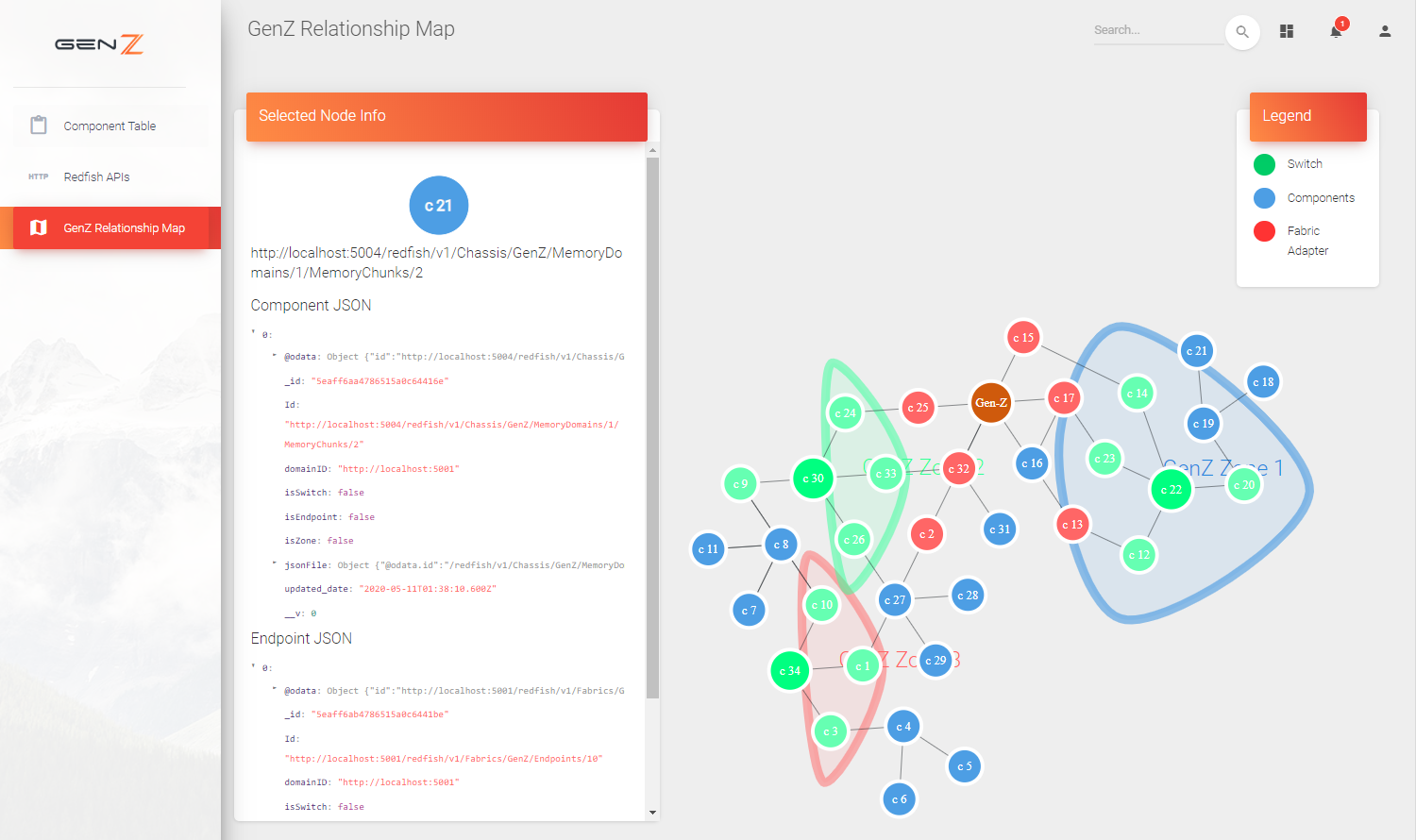
The test serves to evaluate the expected user experience based on the configuration from the simulation data as shown in Figure 6 below (next page) with the data shown in the frontend display. The benchmarks for success are based on a usability standpoint where the connections are clear on the frontend display and a functionality standpoint in that all the components are correctly displayed. It must display all the port connections, zone assignment, and component relations.



**Figure 6: Expected Redfish Simulation for 2D Map Display Testing**

***5.1.2 Result***

The frontend display is shown in Figure 7 below. The map represents the components as nodes. The legend on the right shows the types of components: Switches, Fabric Adapters, and Components. The switch nodes represent the actual switch and the associated switch ports. The Fabric adapters represent fabric adapters that are all connected to the Gen-Z fabric. The components represent various elements which include memory, PCIe devices, and compute systems.

**

**Figure 7: Actual Frontend Display for 2D Map Display Testing**

***5.1.3 Analysis***

The actual map display shows the data in a format more readable for humans as nodes. Clicking through a specific node also shows the component JSON and the Gen-Z endpoint JSON for that component. Therefore, in being able to accurately portray data spanning multiple running instances of simulators, we can show that our project provides a case for usability in IT management utilizing Gen-Z specific information. We can also highlight in our results key categories that can be used for Gen-Z management like the port connections, connected compute systems, fabric managers, and zone assignments. Every component in Figure 6 is properly represented in Figure 7. Additionally, we can compute the total number of nodes that should be on the Gen-Z Relationship Map using equation 1 below. We have 24 total Gen-Z Endpoints across our simulations and 11 switch ports across the three switches. On the frontend, we correctly have 35 nodes displayed.

(1)

***5.1.4 Recommendations***

We recommend continuing the current testing method for the 2D Map as it provides a clear end-to-end representation of the simulation data with the display on the frontend and gives a better idea of how each subsystem works together to bring the result on the 2D display. This testing can also provide a larger idea of the project to work off of for future progress on use cases for Gen-Z manageability as there are a wide variety of Gen-Z configurations displayed in this test mapping.

**6.0 TIME AND COST CONSIDERATIONS**

We had to make multiple design decisions on both the frontend and simulator to fit within our time constraints. Our project accomplished our primary goals of simulating an example Gen-Z system and displaying the data in a human readable format. However, we were unable to have enough time to develop our stretch goals. Reasons for lack of time were due to having to gain familiarity with Gen-Z and Redfish specifications and synchronizing what information needs to be displayed between the simulators and frontend. Additionally, there are a lot of potential directions one could go with simulating Gen-Z behavior. Therefore, we chose the most important for our proof of concept.

**6.1. Learning and Extending the Redfish Simulation**

We had to put in the time to understand how the Redfish specifications and the existing codebase, the Redfish Interface Emulator, worked before we could make the changes we wanted. The Redfish specifications presented a learning curve for us when we wanted to alter the underlying JSON files. For the Redfish Interface Emulator, we experimented with the dynamic behavior, but came to the realization that it would be a lot of work to get the dynamic behavior running and then even more work to make extensions to it. Therefore, we switched to only the static behavior to focus more on building out our example Gen-Z system through the JSON files.

***6.1.1 Learning the Redfish Specification***

Understanding how the JSON files were laid out and what the data represented was crucial to making progress on the simulators. To make any changes to data or create new data for the mockups, there had to be an understanding of what the current data meant and what the new data was supposed to have. For example, in a Gen-Z endpoint, the “Connected Entities” data was crucial to have, and the way it was formatted had to be uniform so it wouldn’t cause issues for the frontend. Gaining familiarity with the Redfish specification in the JSON files presented a learning curve that made progress early much slower. The team members working on the simulators had to put in more time looking at and understanding the JSON files and the Redfish documentation.

***6.1.2 Understanding and extending the Redfish Interface Emulator***

Additionally, time had to be put into understanding the Redfish Interface Emulator codebase and how to build off of it. The dynamic emulation seemed like it would be extremely useful for our project, so we tried to get it running. In doing so, we discovered that the code needed to be updated to get it running in the first place and would need subsequent changes for any extensions we made. Therefore, in the interest of time, we decided to use only the static configurations. Furthermore, the Redfish Interface Emulator codebase is fairly large with many files being called by others. If there was unexpected behavior, debugging took more time because it required looking through code we did not write.

**6.2 Narrowing Overall Functionality**

We narrowed down the overall functionality of our project from having a full set of management features initiated from the management console such as Gen-Z composable systems assignment and dynamic status updates to focus on the 2D mapping showing connected devices over the fabric. The functionality provided by the existing simulator was limited. Expanding the simulator codebase to accommodate for dynamic behaviors was going to cause a setback for the schedule of the sub team working on the simulation. Therefore, we redirected the project functionality to mainly focus on adding and modifying common Gen-Z fields into the Redfish specification for the simulation and reflecting these components to be easily understandable as a 2D Map. This consideration of time and focus helped the group realize the final 2D Gen-Z mapping within schedule. Though the project sidelined functionality that management products support, this project can serve as a baseline to build upon for future Gen-Z management products to support the growing ecosystem for the Gen-Z interconnect.

**7.0 SAFETY AND ETHICAL ASPECTS OF DESIGN**

In the development of our design, security and safety concerns around digital security are important due to the nature of our management proof of concept contributing to the overall ecosystem of IT data centers. These centers could hold sensitive user data such as personal information. Since our management platform will be able to change the states of the connected devices, we must ensure that our protocols can’t be exploited by malicious actors. In order to limit how our management platform can be exploited, we worked in a private repo with our industry sponsor Jon Hass and performed local testing to ensure functionality before releasing any data. Additionally, the safety concerns of managing the Gen-Z devices is important. By accurately monitoring, reporting, and alerting the user of the status of devices and focusing on the correct information to report, we will be able to increase the safety of the underlying topology. Accurately reporting information about the system reduces the risk of major breakdowns of the system. An end-user would be able to diagnose and fix problems as they show up on the console.

We considered looking at legislation around the space of information security since our design was involved in information security. Therefore, we looked at chief legislation for handling data, specifically the General Data Protection Regulation (GDPR), to see the responsibilities that we have in designing a product that is handling data through a controller [5]. Since the GDPR is a data protection regulation that is enforced on any companies handling data in Europe, our project that handles support for a new data interconnect must be aware of the strong security principles that the GDPR and other legislation modeled after it serve to enforce. In the initial stages of implementation, we considered security features like accounts for access in our Management console but found that we needed to prioritize Gen-Z specific management functionality before adding security features like accounts and access privileges. In addition to console features for security, Gen-Z specification holds security features like keys to ensure communication between devices through the Gen-Z fabric is secure. Though our simulation does not reflect these Gen-Z keys for security, our simulation follows the Redfish standard which allows for the Gen-Z specific keys. Any project that builds off of this project must consider the same safety and ethical considerations of handling data.

**8.0 RECOMMENDATIONS**

Moving forward, the next steps for the management tool would be to improve its realism, functionality, and accessibility for all three subsystems. One of the first enhancements would be to turn the simulations into micro services and host the web console and backend all on Dell servers. This would increase accessibility for user testing and allow more people to access our project and give feedback.

In terms of realism, improving the simulators to handle a dynamic setup would be the next steps. Dynamic simulators allow the simulators to change states while running and not having static configurations or statuses. Two ways this could be accomplished by building off the Redfish Interface Emulator dynamic emulation, or by writing scripts that will alter the underlying JSON files which will trigger a reload of the static emulation.

Also, the frontend data displaying design will need to be improved. Since the frontend would now be able to display changed statuses, configurations, and show more live updates, we suggest adding status change alerts and other error or success indicators for the changing behaviors of the simulators. Other specific enhancements for the frontend include adding a physical mapping section to show how the simulators are physically connected in a two-dimensional graph similar to the Gen-Z mapping section. This would show what components are physically located where. Furthermore, adding accounts and permission for different types of users would enhance the security of the website. Allowing only certain users to delete and add API endpoints would ensure safer handling of the data.

**9.0 CONCLUSION**

To contribute to the development of Gen-Z technology, we created a proof of concept Gen-Z management console that displays data from a simulated Gen-Z system. We created three subsystems: a console frontend with two views, a backend with an attached MongoDB database, and a simulated Gen-Z system. The console frontend has a component view, where every element of the simulator can be viewed and provides easy navigation through the simulators. The backend is a connection between the frontend and simulators and allows for asynchronous communication between the other two subsystems. It also contains a MongoDB database that allows for detecting changes in simulator state and retaining information in the event of a simulator crashing. We developed a Python script to run multiple simulators locally over LAN connections. The simulators are built off of the Redfish Interface Emulator codebase, which is a Python Flask application allowing for Redfish HTTP requests. Each simulator represents a separate device or enclosure to better simulate a real system. The underlying system data is represented in JSON files in a directory structure that adheres to the Gen-Z Redfish standards. The component view of the frontend allows for seeing all the details, while the Gen-Z relationship mapping allows for a visual representation of the connections over the Gen-Z fabric. Both of the views were end-to-end tested to make sure the data in the simulators was displayed properly on the frontend. We recommend the next steps for the project should be to put the simulators in micro services and host both the backend and the frontend on servers to increase accessibility of the system. The simulators should be extended to allow for dynamic behavior so that it can change states while running and allow for the frontend to display that change. In terms of security, the website could create permissions so that only certain people can add, delete, or change data from the frontend. By accurately modelling Gen-Z networks and displaying information of its devices, our Gen-Z manageability proof of concept provides a starting framework for a working manageability console that will help an end-user better understand and maintain a Gen-Z system.

**REFERENCES**

[1] “Z Technology: Gen-Z Consortium,” *Gen-Z*. [Online]. Available: <https://genzconsortium.org/about-us/gen-z-technology/>

[2] “Redfish,” *DMTF*. [Online]. Available: <https://www.dmtf.org/standards/redfish>

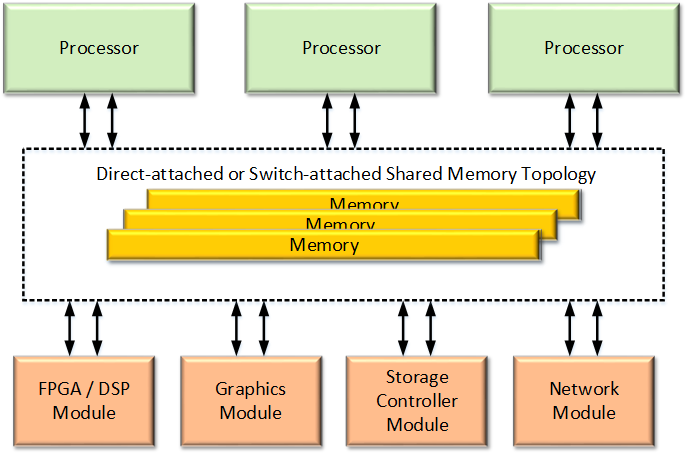
[3] “Introduction - NEO v2.5 - Mellanox Docs.” [Online]. Available: <https://docs.mellanox.com/display/NEOv25/Introduction>

[4] A. Pathan, “United States Patent: 10462020 - Network device user interface,” 10462020, 29-Oct-2019

[5] “Chapter 4 – Controller and processor,” *General Data Protection Regulation (GDPR)*. <https://gdpr-info.eu/chapter-4/> (accessed May 03, 2020)

**APPENDIX A – RELEVANT STANDARDS**

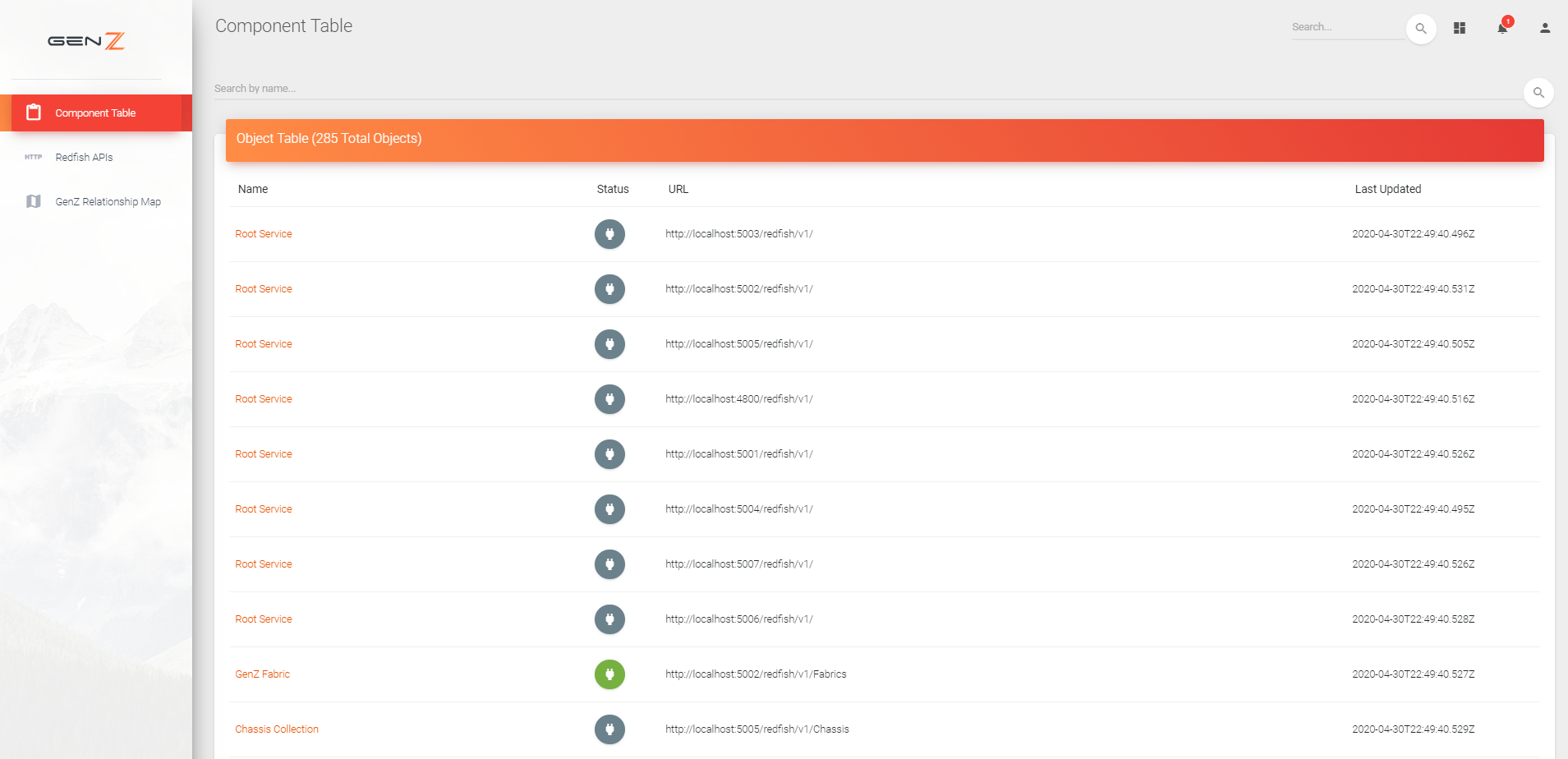
**APPENDIX A – RELEVANT STANDARDS**

**Figure 1. Gen-Z Memory-Centric Architecture** [1]

This figure demonstrates the relevant architecture of the Gen-Z system. The traditional computer architecture has each processor with its own local memory. If a processor is disconnected, then local memory is also gone. The topology of Gen-Z shares memory which allows the memory component to be independent of processors. It also allows for dynamic switching of how the memory is used and assigned. The entire system can have the memory shared between all the module

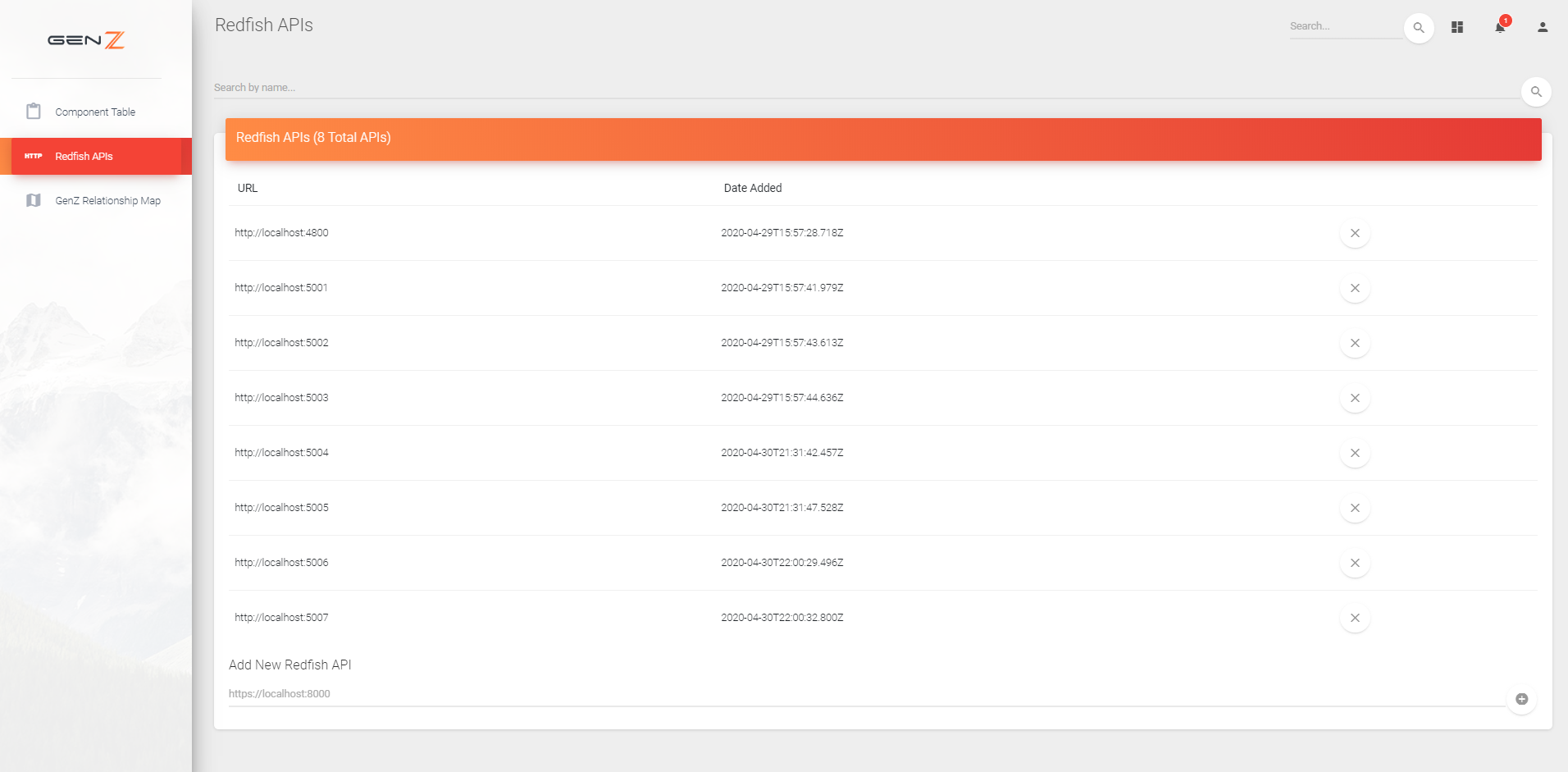
**APPENDIX B - FRONTEND FIGURES**

**APPENDIX B - FRONTEND FIGURES**



**Figure 3. Frontend Component Table**

The figure shows how the Component Table on the frontend is displayed. There is a search bar at the top to search for the name of the component. Below is the full list of all objects in the simulation. It displays the name, status, URL of each item, and last updated time. The status is color coded so that green means enabled, red means an error state such as disabled, and grey means that the data shows an enumeration of the states or an unexpected format. Since we were not focused on simulating changing behavior, we left most of the states as is. A lot of the data had all the states enumerated, which is why a lot of the statuses are grey. Clicking on any of the rows will bring up a new page with the full details for that component along with clickable links up the file hierarchy and to connected components.



**Figure 4. Frontend Redfish API Table**

This figure displays all the Unique Resource Identifiers (URIs) for the simulators. A URI can be deleted by pressing the X button on the right-hand side of that URI’s row. One can be added at the bottom by typing in the path. That triggers the database to recursively add all data associated with that URI.